Preprint
UCRL-JC-135886

Improving the Extension Facilities in C+

P.F. Dubois and B.A. Scott

This article was submitted to
8th International Python Conference
Alexandria, VA
January 24-27, 2000

September 24, 1999

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

September 24, 1999

Improving the Extension Facilities in C+

Paul F. Dubois
Program for Climate Model Diagnosis and Intercomparison
Lawrence Livermore National Laboratory[1]
dubois1@llnl.gov

Barry A. Scott
barry@scottb.demon.co.uk

Eighth International Python Conference
January, 2000
Washington, D.C.

Abstract

CXX is a facility for extending Python using C++. Recently, the authors have substantially revised and improved the way in which you create extension objects and extension modules in C++. The method is now much more natural and has much less overhead, both in the code generated and in the effort needed to create the objects and extensions.

1.0 Introduction to climate modeling

1.1 CXX

CXX is a package of header files and the supporting implementations for a series of classes and auxilliary functions to assist the user in writing Python extensions in C++. CXX is written in standard C++. The source code is available for free redistribution as a beta release as part of the LLNL Python extensions distribution, which is available at **ftp://ftp-icf.llnl.gov/pub/python/LLNLDistribution.tgz**. Please see the legal notices enclosed in that package.

1.2 Recent developments in CXX

The idea behind CXX, as described in the paper presented last year [2], is twofold. First, the C API for Python is encapsulated in some classes such as Object, Dict, and Tuple, so as to greatly simplify the use of these objects in compiled code. For example, the following CXX code creates a dictionary and adds two named (Python) integers to it:

```
Dict d;
d ["one"] = Int (1);
d ["two"] = Int (2);
```

The CXX class hierarchy for this portion is shown in Table 1.

TABLE 1. CXX Class Heirarchy

- Object
  - Type
  - Module
  - Integer
  - Float
  - Long
  - Complex
  - Char (Strings of length 1)
  - SeqBase<T>
    - Sequence (= SeqBase<Object>)
    - String
    - Tuple
    - List
    - Array (NumPy array)
  - MapBase<T>
    - Mapping (= MapBase<Object>)
    - Dict
- Exception
  - StandardError
    - IndexError
    - RuntimeError
  - ... (more classes corresponding to the Python exception heirarchy).

In addition there are a number of functions defined at the global (namespace Py) level. These include the usual binary arithmetic operators and stream output operators.

The second part of CXX was more experimental; it was an attempt to make it easier to construct your own objects and extension modules, without having to construct odd-looking tables and use mysterious non-standard constructions such as the infamous "staticforward" declaration. Barry Scott has substantially revised this section of CXX with some assistance from Paul Dubois.

One caution for those who are not very experienced at C++. This work uses some advanced C++ techniques, such as templates and inheriting from a class templated upon the inheriting class. Such readers will should simply try to absorb the gestalt rather than the details. As a user, advanced techniques are not required; indeed, the purpose of the work is to reduce the intellectual load on the user, not to increase it.

During the period since the Seventh International Python Conference, C++ compiler inadequacies have largely disappeared. In particular, CXX can be used with the free Gnu project C++ compiler, g++. Better performance is still obtainable with compilers such as the Kuck and Associates KAI compiler, however.

2.0 Constructing an extension object

PythonExtension is a class from which you inherit to create a new Python extension object. You override behaviors of this class to define Python methods such as repr(), and add methods to the class that are callable from Python as methods on instances of your new class.

Here, for example, is the interface to an extension object, class "r", which resembles a "range" object. CXX's constructs are in the namespace "Py", so you can identify them in what follows because they are preceded by "Py::".

```
class r: public Py::PythonExtension<r> {
public:
  long start;
  long stop;
  long step;
  r (long start_, long stop_, long step_ = 1L);
  virtual ~r()
  static void init_type(void); // see discussion below
  long length() const ;
  long item(int i) const ;
  r* slice(int i, int j) const ;
  r* extend(int k) const;
  STD::string asString() const ;

  // override functions from PythonExtension
  virtual Py::Object repr();
  virtual Py::Object getattr( const char *name );
  virtual int sequence_length();
  virtual Py::Object sequence_item( int i );
  virtual Py::Object sequence_concat( const Py::Object &j );
  virtual Py::Object sequence_slice( int i, int j );

  // define python methods of this object
  Py::Object amethod (const Py::Tuple& args);
  Py::Object value (const Py::Tuple& args);
  Py::Object assign (const Py::Tuple& args);
  Py::Object reference_count (const Py::Tuple& args) ;
  };
```

Note the inheritance of r from Py::PythonExtension<r>. We then override the default behaviors of the object as desired, for example by defining a method "sequence_item" to calculate the reaction of an r object to integer subscripting:

```
Py::Object r::sequence_item(int i)
{
  return Int(item(i));
}
```

Methods such as "amethod" receive as their arguments the Tuple of arguments passed to Python, and return an Object. Here amethod returns a list consisting of itself and its one argument:

```
Py::Object r::amethod (const Py::Tuple &t )
{
        t.verify_length(1); // check there is just one argument
        Py::List result;
        result.append(Py::Object(this));
        result.append(t[0]);
        return result;
}
```

The connection between Python and r is completed by the addition of a constructor method to one of our Python extension modules, and by the static routine r::init_type(). Here is that static method from class r:

```
void r::init_type()
{
        behaviors().name("r");
        behaviors().doc("r objects: start, stop, step");
        behaviors().supportRepr();
        behaviors().supportGetattr();
        behaviors().supportSequenceType();

        add_varargs_method("amethod", &r::amethod,
                "demonstrate how to document amethod");
        add_varargs_method("assign", &r::assign);
        add_varargs_method("value", &r::value);
        add_varargs_method("reference_count", &r::reference_count);
}
```

3.0 Constructing Python extension modules

Likewise, the construction of a Python module has been simplified: You create a clas inheriting from ExtensionModule, and the methods of the class become the Python methods. You also can add items to the module dictionary.

```
class example_module : public ExtensionModule<example_module>
{
public:
        example_module()
                : ExtensionModule<example_module>( "example" )
        {
                r::init_type();  // initialize the "r" type discussed above
                add_varargs_method("sum", ex_sum,
                                "sum(arglist) = sum of arguments");
                add_varargs_method("test", ex_test,
                                "test(arglist) runs a test suite");
                add_varargs_method("r", new_r,
                                "r(start,stop,stride)");

                initialize( "documentation for the example module" );

                Dict d( moduleDictionary() );
                d["a_constant"] = Float(3.14159);  // add a famous constant
                                                   // to this module
        }

        virtual ~example_module() {}

private:
        Object new_r (const Tuple &rargs)
        {
                if (rargs.length() < 2 || rargs.length() > 3)
                {
                        throw RuntimeError(
                                "Incorrect # of args to r(start,stop [,step]).");
                Int start(rargs[0]);
                Int stop(rargs[1]);
                Int step(1);
                if (rargs.length() == 3)
                {
                        step = rargs[2];
                }
                if (long(start) > long(stop) + 1 || long(step) == 0)
                {
                        throw RuntimeError("Bad arguments to r(start,stop [,step]).");
                }
                return asObject(new r(start, stop, step));
        }
        Object ex_sum (const Tuple &a)
        {
                Float f(0.0);
                for( int i = 0; i < a.length(); i++ )
                {
                        f = f + Float(a[i]);
                }
                        return f;
        }

        Object ex_test( const Tuple &a)  { implementation omitted }
}
```

Note the support provided by the CXX machinery. For example, if one of the arguments to "sum" is not a Python floating-point number, an exception is thrown, any temporary objects such as f are cleaned up, and a Python exception results.

Now we add the initialization routine Python requires, which now must only construct a permanent instance of our module class:

```
void initexample()
{
        static example_module *example = new example_module;
}
```

The examples presented here have been somewhat simplified from those in the CXX package for expository purposes. Potential users should consult the files in the Demo directory for further examples.

4.0 Summary

A fully object-oriented approach, in which extension objects are constructed by inheritance and the overriding of methods controlling the objects' behavior, makes for a natural C++ approach. Likewise, extension modules as classes whose methods become available from Python seems to us to be natural and easy.

5.0 References

1. This work was produced in part at the University of California, Lawrence Livermore National Laboratory (UC LLNL) under contract no. W-7405-ENG-48 (Contract 48) between the U.S. Department of Energy (DOE) and The Regents of the University of California (University) for the operation of UC LLNL. The views and opinions of the authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.
2. P. F. Dubois, "A facility for extending Python in C++", in *Proceedings of the Seventh Internation Python Conference*, Foretec Seminars, Reston,VA, 1998. pp. 61-68.